Enabling Component-based Mobile Cloud Computing with the AIOLOS Middleware

Steven Bohez
Department of Information Technology
Ghent University - iMinds
Gaston Crommenlaan 8/201
B-9050 Ghent, Belgium
steven.bohez@ugent.be

Elias De Coninck
Department of Information Technology
Ghent University - iMinds
Gaston Crommenlaan 8/201
B-9050 Ghent, Belgium
elias.deconinck@ugent.be

Tim Verbelen
Department of Information Technology
Ghent University - iMinds
Gaston Crommenlaan 8/201
B-9050 Ghent, Belgium
tim.verbelen@ugent.be

Pieter Simoens
Department of Industrial Technology and Construction
Ghent University
Valentin Vaerwyckweg 1
B-9000 Ghent, Belgium
pieter.simoens@ugent.be

Bart Dhoedt
Department of Information Technology
Ghent University - iMinds
Gaston Crommenlaan 8/201
B-9050 Ghent, Belgium
bart.dhoedt@ugent.be

ABSTRACT

Currently, mobile and wearable devices (such as smartphones and tablets) and cloud computing are converging in the new, rapidly growing field of mobile cloud computing. Emerging distributed cloud architectures such as edge clouds can be used to support and scale out resource-intensive, low-latency mobile applications. However, at the moment, a lot of burden is put on the application developer in order to develop and deploy distributed cloud-enabled mobile applications. Therefore, we present AIOLOS: an integrated middleware platform that supports transparent distributed deployment and scaling among mobile devices and cloud infrastructures. To evaluate the middleware, we show experimental results of AIOLOS using a complex 3D mapping use case.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Design, Algorithms, Management

Keywords

Mobile cloud computing, Middleware, OSGi

ARM'14 December 9, 2014, Bordeaux, France

DOI: http://dx.doi.org/10.1145/2677017.2677019

1. INTRODUCTION

Today, mobile devices are among the fastest growing market segments, with billions of smartphones and tablets to be sold in 2014[1]. As these devices are portable and always-connected, they are becoming the preferred platform for a myriad of applications. Current developments in hardware miniaturization and near-to-eye display technologies open the door to a new class of wearable devices such as smart watches and glasses. However, despite increasing hardware capabilities, these wearable devices will always be resource poor compared to fixed hardware, due to the intrinsic limitations concerning weight, size, battery and heat dissipation.

In order to mitigate the hardware limitations on mobile devices, cloud computing is often seen as a silver bullet, allowing users to use remote infrastructure in an on-demand fashion [4]. Moreover, these cloud systems provide elastic horizontal scaling, allowing to easily handle peak loads. However, high and variable latency, as well as bandwidth constraints limit the applicability of the cloud for mobile computing, especially when it comes to media processing [15]. Therefore, mobile cloud computing is evolving more and more towards an edge cloud architecture, with highly distributed infrastructure deployed closer to the access network of the end user [10]. Extensive research has covered the problem of dynamically offloading computation from the mobile device to the (edge) cloud in order to improve user experience or energy consumption. The focus, however, generally lies on personalized, single-user applications. Such applications hardly benefit from the elasticity offered by the cloud as a single user can only generate so much load. In our work, we explicitly take into account multi-user applications, which experience significant shifts in load and prove more of a challenge to scale out cloud resources effectively.

From a developer perspective, deploying an application on the mobile cloud means that one has to take into account a distributed ecosystem on which the application or different parts thereof should be deployed, as illustrated on Figure 1.

[1] `http://www.gartner.com/newsroom/id/2692318`

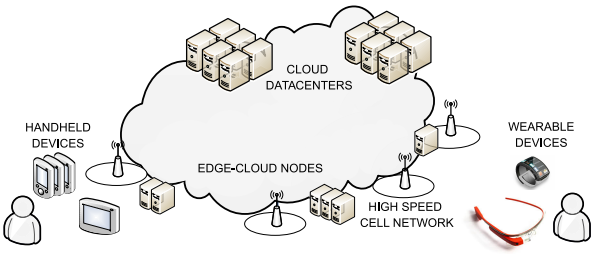

Figure 1: The mobile cloud computing ecosystem targeted by AIOLOS middleware consists of back-end cloud datacenters and edge cloud compute nodes to support applications for mobile and wearable devices.

This puts a huge burden on the application developer, as he is responsible for supporting one or a number of different mobile and cloud platforms. Coping with the multitude of vendors and device generations has proven to be a major issue in mobile cloud computing [14]. Moreover, the developer has to make the application distribution aware (i.e. supporting a remote method call mechanism such as web services), and configure scaling rules.

To address these issues, we present AIOLOS[2] [22], a middleware platform for mobile cloud computing. This platform enables easy development of component-based applications, for which components can be transparently deployed and scaled on back-end cloud datacenters, on edge-cloud nodes or locally on a mobile device. By managing applications on a component level, AIOLOS is able to (i) offload compute-intensive parts of the application from the mobile device to the (edge-)cloud, (ii) monitor application components and scale out in the cloud on a fine grained component level and (iii) quickly reprovision cloud Virtual Machines (VMs) by stopping, starting or migrating components. Moreover, AIOLOS features scaling on both the infrastructure and application level when deployed on an Infrastructure as a Service (IaaS) cloud, effectively turning it into a Platform as a Service (PaaS) cloud. This allows the developer to focus on application functionality, rather than deployment, distribution and scaling, which is handled by the framework.

Whereas in [22] we focus on the algorithms in AIOLOS used for offloading from a single device to a single server, this paper focuses on the inner workings of AIOLOS itself as well as its ability to scale out multi-user applications on a multi-tier infrastructure. As a use case we present Mercator, a complex 3D mapping application, and show how AIOLOS can automatically scale out its constituting components to handle a growing user count.

2. AIOLOS OVERVIEW

The main objective of the AIOLOS framework is to decouple (distributed) application deployment from application development. In order to be able to transparently distribute parts of the application, we adopt a component-based application model. We assume an application is built as a number of loosely coupled software components, that communicate through well defined service interfaces, as depicted on Figure 2. The biggest advantage of the component-based approach is that the application developer can solely focus on application functionality by implementing the service interfaces, without having to bother on how the components will actually communicate on a distributed infrastructure.

An example deployment of this application is shown in Figure 3. Each infrastructure node runs the AIOLOS middleware. This could be embedded in a mobile application on a wearable device, or running inside a VM on top of an IaaS cloud. In order to transparently distribute application components, AIOLOS creates proxies for all component service interfaces, hiding the actual component implementations. The proxy can forward method calls to local instances (represented with a continuous line), or execute a remote method call (shown as a dashed line). When multiple (remote) component instances are available, different proxy policies can be used to forward the call. For example in case of component B, the proxy on the mobile device can switch between local and remote execution, e.g. to minimize the execution time or the energy spent. In the case of component E, this mechanism can be used to scale out a component to the cloud, and use a load balancing policy to distribute load among the available instances.

Because each service call between two application components passes through an AIOLOS proxy, we can also gather monitoring information about all inter-component communication. For each method call the size of the arguments and return value is recorded, as well as the execution time of the call. This monitoring information can be used to build a runtime model of the application, and to identify candidate components to offload or to scale out.

As the application components are loosely coupled, they can be deployed on any of the AIOLOS nodes in the network. Application components can be started, stopped and migrated at runtime, allowing to reconfigure the application deployment at runtime, and enabling quick reprovisioning of running VMs in the cloud.

3. DESIGN AND IMPLEMENTATION

As the component-based concept is central in our approach, we have built AIOLOS on top of OSGi [18]. The OSGi core specification defines a service oriented module management system for Java, allowing to dynamically load and unload software modules – called bundles – at runtime. OSGi uses the whiteboard pattern allowing components to register and lookup service interface implementations. The portability of Java enables the execution of the same code on different platforms and architectures, supporting both server machines and mobile devices running Android. The AIOLOS middleware runs on top of OSGi, and is by itself also designed as a number of components. The internal design of the AIOLOS framework is depicted in Figure 4, and

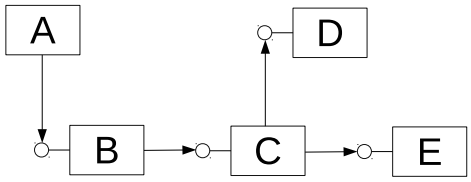

Figure 2: Applications are composed of loosely coupled software components that communicate through well defined service interfaces.

[2]Source available at `http://aiolos.intec.ugent.be`

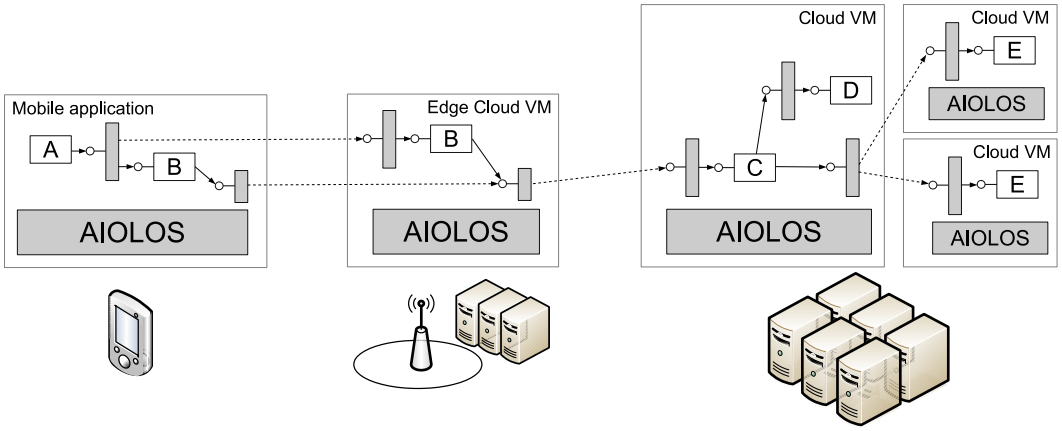

Figure 3: The AIOLOS platform can run both on mobile devices and (edge) cloud VMs. Every service interface is proxied by AIOLOS, allowing to transparently switch between local and remote execution (i.e. B), or scale out in a cloud environment (i.e. E).

is structured in three layers: the core, monitoring and deployment layer.

The core layer provides the basic building blocks enabling service distribution and proxying. The **RemoteServiceAdmin** implements a remote procedure call protocol to forward method calls to remote component instances. The protocol is based on R-OSGi [13], and uses Kryo[3] for fast and efficient object graph serialization. In order to remain compatible with the OSGi specification, the RemoteServiceAdmin implements the interfaces specified in the OSGi Enterprise Release [19] regarding remote services. Different AIOLOS nodes are interconnected via their **TopologyManagers**, which will exchange the available services on the different nodes, allowing to lookup and connect to remote service instances. Every service interface is proxied by the **ProxyManager**, hiding the actual service implementation. When more than one service instance is available, a ProxyPolicy is used to determine to which instance service calls should be forwarded. Different policies are available, for example a round robin policy for load balancing in the cloud, or a decision algorithm based on the method argument size for mobile offloading as proposed in [22].

[3] `https://github.com/EsotericSoftware/kryo`

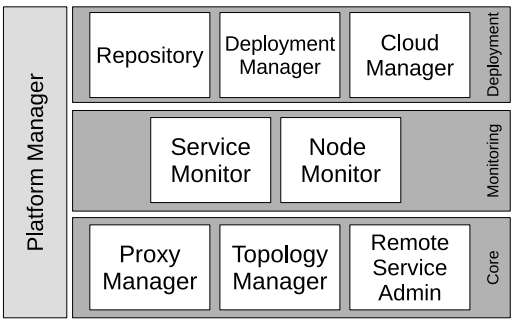

Figure 4: The AIOLOS middleware can be split up into three layers: the core, monitoring and deployment layer. The PlatformManager provides a unified interface to interact with all AIOLOS components.

The monitoring layer collects information on service level and node level. The **ServiceMonitor** receives callbacks from the ProxyManager each time a service method is called, and records the argument size, return value size and execution time of each method. This allows to identify resource-intensive components that should be offloaded or scaled out. The **NodeMonitor** operates on the level of the operating system, collecting metrics such as CPU usage, memory consumption, etc. The current implementation is restricted to Linux based systems by reading the /proc/ filesystem. These metrics can be used to identify overloaded nodes from which components should be migrated.

The deployment layer provides components to lookup and deploy components, and to start or stop new VMs in a cloud environment for scaling out. All deployment artefacts (i.e. .jar files) are kept in a repository. The **Repository** component provides a searchable index of all artefacts and their capabilities. Just like the RemoteServiceAdmin, our implementation follows the Repository specification from the OSGi Enterprise Release. The **DeploymentManager** provides an interface to start, stop or migrate components on an AIOLOS node. Components are fetched from available Repositories, and using the OSGi capability model also component dependencies can be resolved at runtime. When running on a cloud environment, the **CloudManager** can be used to start or stop new VMs. These VMs are then automatically provisioned with AIOLOS running on top of an OSGi runtime, and can be used to scale out application components. In order to support multiple cloud providers, our CloudManager implementation uses Apache jclouds[4].

Finally, the **PlatformManager** offers a unified interface to all layers. This interface can for example be used for creating an autonomic decision engine, implementing a feedback loop gathering monitoring information and executing new deployment decisions based thereon. Application developers can also use this interface to collect and inspect monitoring information of their application, or to provide their own ProxyPolicies and implement application-specific scaling behaviour.

[4] `http://jclouds.apache.org`

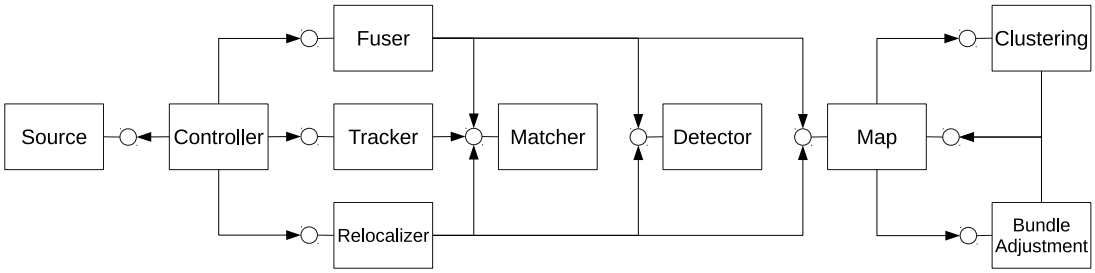

Figure 5: Overview of the Mercator architecture. Videoframes are fetched from the video source, after which features are detected and matched in order to add them to the Map. The Map then self-optimizes using bundle adjustment and clustering.

4. USE CASE: MERCATOR

To illustrate the features of AIOLOS, we implemented a prototype of Mercator [17], a proposed application for online and scalable 3D mapping. By processing live video feeds of mobile devices such as smart glasses or phones using a Structure from Motion (SfM) pipeline, a detailed three-dimensional map of the users' environment is created. SfM is a computer vision technique that allows to reconstruct a 3D model without any prior information about the subject to be reconstructed or the position of the camera capturing it. The goal of Mercator is that a large group of users may collaboratively expand this model of their environment, eventually resulting in a crowd-sourced 3D map of the world. Such a 3D map would enable a wide range of applications, such as Augmented Reality (AR) and indoor navigation. While other attempts have recently been made to create a scalable 3D reconstruction pipeline [1], Mercator distinguishes itself by providing online mapping performed by multiple users.

Figure 5 shows an overview of the different components making up Mercator. To build a 3D model, the **Controller** first fetches frames from the **Source**, e.g. a mobile device's physical camera or a prerecorded video file. These are then forwarded to the **Tracker**, which will estimate the position and orientation of the camera. This pose estimation is performed by detecting and matching 2D image features with known 3D features points of the already reconstructed model. Detecting image features is performed by the **Detector** service, while feature matching is done by the **Matcher**. Note that multiple Source components require multiple Controller and Tracker components. The **Map** service stores (a part of) the reconstructed model in the form of a point cloud of 3D features along with the frames in which they were detected. A single Map service can be shared between multiple users, allowing them to collaboratively expand the 3D model.

When a client's Controller has sufficient confidence in the Tracker's pose estimation, it forwards frames to the **Fuser**. The Fuser will estimate the 3D position of newly detected features in the frame and add them to the Map, along with the frame and its estimated pose themselves. The Fuser will also refine the position of existing 3D features. If the tracking quality is poor, which may be caused by e.g. rapid camera movement, the Controller will ask the **Relocalizer** to make a rough estimate of the camera position by looking for similarities with previously processed frames. When new frames and 3D feature points are added to the sparse point cloud, their pose and position need to be optimized with respect to the rest of the model. In order to do this, the Map calls the **BundleAdjustment** service. Bundle adjustment [20] (not to be confused with OSGi bundles) performs a joint refinement of frame poses and feature point locations by nonlinear minimization of the reprojection error.

However, the execution times of modifying or querying the model dramatically increase with the extent of the model. This is especially the case for bundle adjustment, for which the execution time scales cubically in the number of feature points and frame poses. To cope with this, a **Clustering** service will scale the Map service by splitting the model into several overlapping partial models, each covering a local geographic area. Each partial model can then be updated and optimized independently of the others, which in turn allows the other services to scale out to match the increased demand. To further reduce execution times, the Clustering service will perform a filtering operation on the model following bundle adjustment. 3D features points that likely belong to the same physical point in space are consolidated and redundant frames are removed.

Mercator is a quite complex use case. Each mobile client requires a unique instance of the Source, Controller & Tracker components, as they maintain session state. Likewise, the Map component can not be scaled out by simply replicating the service but requires a specialized clustering algorithm. The remainder of the components however (Fuser, Relocalizer, Dectector, Matcher, Clustering & BundleAdjustment), are stateless and can automatically be scaled out by the AIOLOS framework in order to cope with the demand. As more users participate, more frames need to be processed by the Fuser, which in turn expands the point cloud in the Map. This triggers the clustering algorithm, starting more Map instances which require filtering and bundle adjustment.

5. EXPERIMENTAL RESULTS

We show the feasibility of our approach by experimenting with the Mercator application on top of AIOLOS. Experiments are performed on top of an OpenStack Icehouse[5] IaaS cloud consisting of four compute nodes equipped with an Intel Xeon E5-2620 processor and 16GB RAM. A single management VM controls the experiment by booting the required VMs. Clients are simulated by a client VM provisioned with a Source, Tracker and Controller component instance that will process an identical pre-recorded video. This setup is used instead of actual mobile devices in order to facilitate experimenting, but results in the same load pat-

[5] `http://www.openstack.org`

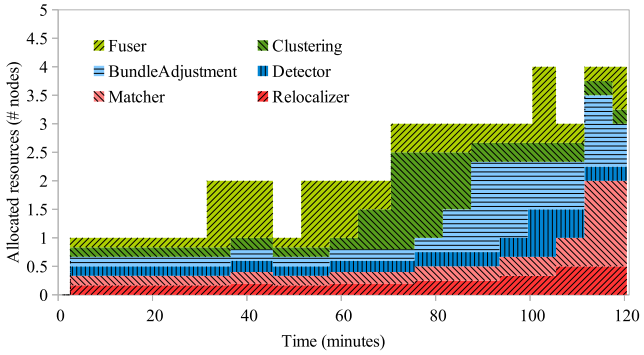

Figure 6: The allocated resources for each of the component types in the Mercator use case for the duration of the experiment. Resources are measured in the fractions of worker VMs available.

tern if camera and video resolution match. Initially, a single worker VM is booted that provides the remaining services required by the clients as well as a Map service. The management VM also processes the monitor information gathered by the NodeMonitors and ServiceMonitors and decides which components to scale up or down in a decision loop.

A scaling policy based on CPU load thresholds is used that scales on a component level. More advanced scaling policies incorporating other resource limitations are also possible, but are not the focus of this paper. The scale-up procedure is triggered when the CPU load on a worker VM, averaged over the monitor interval, exceeds 75%. Scaling up is performed by starting an additional instance of the component consuming the most CPU time on that specific worker. A new worker VM is booted when all existing worker VMs already provide an instance of that component. Similarly, when load drops below 50%, components are scaled down by stopping the component consuming the least CPU time. If that component happens to be the only instance, however, it is migrated to another available node. If all components on a worker VM have been stopped, the VM itself is shut down as well. Multiple components can be scaled up or down in each iteration of the decision loop, depending on how many nodes are over- and underloaded. It may occur that the same component needs to scale up for one node and down for another in a single iteration, in which case the component on the overloaded node is stopped.

Starting from a single, empty Map, a new client is added every 15 minutes until a total of 5 clients are simultaneously expanding the model. The first client arrives after 15 minutes and the final one after 75 minutes. The scaling loop monitors over an interval of 3 minutes, which is just long enough to let all method calls be balanced among the available workers. Figure 6 shows the allocated resources for each of the components and how the component-based scaling policy reacts to the increasing load. For each of the six components, the allocated resources are calculated as the sum of $1/\#component(n)$ over all worker VMs n where the component is instantiated, where $\#component$ is the number of components present on n.

We observe that as load increases, so does the number of worker VMs, to a total of four after two hours. Moreover, the allocated resources vary between components. Initially, all components get 1/6 of the first worker VM. However, as more clients are fusing frames in the model, the Fuser gets allocated more resources (i.e. a new worker is booted on which a Fuser component is instantiated) to handle the increased load. Downscaling of the Fuser occurs because one of the users loses track of its position and takes some time to have regain sufficient confidence to expand the model. After an hour of expanding the model, the filtering performed by the Clustering component starts to take more time and requires an additional VM. After about 80 minutes, the model gets clustered into several submodels, each requiring bundle adjustment to be performed. The platform reacts by downscaling the Clustering component in favor of upscaling the BundleAdjustment component. Approaching the two hour mark, a fourth worker VM gets booted to upscale the Matcher, which is now called more often due to the increased size of the model. During the course of the experiment, starting a new VM took on average (standard deviation) about 70 s (12 s), while starting an individual component on an already booted VM only took about 1.8 s (1.1 s). This clearly illustrates the benefits of a component-based approach in the fast reprovisioning of applications.

6. RELATED WORK

An important aspect of mobile cloud computing is partitioning mobile applications and offloading parts thereof to remote infrastructure, also denoted as cyber foraging [11]. Different approaches exist, offloading parts of the application either using a method (Scavenger [7], MAUI [6]) component (OSGi in [8], Sprout in Odessa [12], weblets in [23]) or VM (CloneCloud [5], COMET [9]) level granularity. Using software components as unit of deployment rather than methods results in less fine-grained optimization options, but reduces the overhead of monitoring and distributing. A more detailed survey on cyber foraging can be found in [16]. Although these systems focus on application partitioning and offloading, none of them actually tackle application scaling at the cloud side. Therefore, AIOLOS takes into account scaling mechanisms in order not only to offload parts of the application, but also scale them in the cloud to process all client-side requests.

Besides distant cloud infrastructure, applications can also be offloaded to resources nearby. Satyanarayanan et al. [15] propose the use of cloudlets, consisting of trusted infrastructure deployed in the local area network, mitigating high network latency. Bonomi et al. introduce the concept of fog computing [3], extending traditional cloud computing with a largely distributed number of nodes deployed at the edge of the network. This shows that the future Internet will consist not only of a massive number of client devices backed by a few large datacenters, but also a lot of intermediate infrastructure will be deployed at the edge. Therefore, our platform is designed to be able to cope with a various number of deployment points.

In order to scale in the cloud, new VMs have to be provisioned at runtime. Current provisioning systems such as Chef or Puppet provide tools to manage the infrastructure, relying on custom scripting and management tools [2]. Although higher level management tools exist [21], these tools allow to configure a VM at boot time, but do not allow to reconfigure the machine at runtime. To mitigate this shortcoming, our framework operates at a component level, which allows to quickly reconfigure the VM by (un)installing components at runtime. PaaS providers such as Google App

Engine[6] do hide the provisioning complexity from the end user, but then again this offers less control over the platform, and requires a fixed client-server separation of the application.

7. CONCLUSION AND FUTURE WORK

In this paper, we described the inner workings of AIOLOS, a middleware for supporting component-based applications in mobile cloud environments. AIOLOS allows to transparently distribute applications on a variety of mobile devices and fixed infrastructure. By being able to start or migrate components at runtime, computation can be easily offloaded to (edge) cloud servers and applications scaled out to match demand. Moreover, when deployed on an IaaS-platform, AIOLOS can scale both on an application level as well as an infrastructure level by booting and provisioning additional VMs. Finally, AIOLOS provides an API that allows developers to gather monitoring information and define their own offloading and scaling policies. The possibilities of AIOLOS were illustrated using the Mercator use case, an application for scalable, online 3D mapping. Our experiment showed that a simple scaling policy is able to use the benefits of both infrastructure level scaling as well as fast reprovisioning of components to handle the demands of Mercator.

In future work, we look to expand support for other operating systems and IaaS-providers as well as develop more advanced (distributed) scaling algorithms that improve resource usage and stability.

8. ACKNOWLEDGMENTS

Steven Bohez is funded by Ph.D. grant of the Agency for Innovation by Science and Technology in Flanders (IWT). Tim Verbelen is funded by Ph.D. grant of the Fund for Scientific Research, Flanders (FWO-V). This project was partly funded by the UGent BOF-GOA project "Autonomic Networked Multimedia Systems" and by the FWO-V project "SPEC: Intelligent SuPer-Elastic Clouds".

9. REFERENCES

[1] S. Agarwal et al. Building Rome in a day. *Comm. of the ACM*, 54(10):105, 2011.

[2] R. Barrett et al. Field studies of computer system administrators: Analysis of system management tools and practices. In *2004 ACM Conf. on Computer Supported Cooperative Work, Proc.*, pages 388–395. ACM, 2004.

[3] F. Bonomi et al. Fog computing and its role in the internet of things. In *1st Edition of the MCC Workshop on Mobile Cloud Computing, Proc.*, pages 13–16. ACM, 2012.

[4] R. Buyya et al. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009.

[5] B.-G. Chun et al. CloneCloud: elastic execution between mobile device and cloud. In *6th Conf. on Computer Systems, Proc.*, page 301. ACM, 2011.

[6] E. Cuervo et al. MAUI: making smartphones last longer with code offload. In *8th Int. Conf. on Mobile Systems, Applications, and Services, Proc.*, page 49. ACM, 2010.

[7] M. Daro Kristensen. Scavenger: Transparent development of efficient cyber foraging applications. In *2010 IEEE Int. Conf. on Pervasive Computing and Communications, Proc.*, pages 217–226. IEEE, 2010.

[8] I. Giurgiu et al. Dynamic software deployment from clouds to mobile devices. In *13th Int. Conf. on Middleware, Proc.*, pages 394–414. Springer, 2012.

[9] M. S. Gordon et al. Comet: Code offload by migrating execution transparently. In *10th USENIX Conf. on Operating Systems Design and Implementation, Proc.*, pages 93–106. USENIX Association, 2012.

[10] S. Islam et al. Giving users an edge: A flexible cloud model and its application for multimedia. *Future Generation Computer Systems*, 28(6):823–832, 2012.

[11] D. Kovachev et al. Mobile Cloud Computing: A Comparison of Application Models. *CoRR*, abs/1107.4, 2011.

[12] M.-R. Ra et al. Odessa: enabling interactive perception applications on mobile devices. In *9th Int. Conf. on Mobile Systems, Applications, and Services, Proc.*, page 43. ACM, 2011.

[13] J. S. Rellermeyer et al. R-osgi: Distributed applications through software modularization. In *ACM/IFIP/USENIX 2007 Int. Conf. on Middleware, Proc.*, pages 1–20. Springer, 2007.

[14] Z. Sanaei et al. Heterogeneity in Mobile Cloud Computing: Taxonomy and Open Challenges. *IEEE Comm. Surveys & Tutorials*, 16(1):369–392, 2014.

[15] M. Satyanarayanan et al. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[16] M. Sharifi et al. A Survey and Taxonomy of Cyber Foraging of Mobile Devices. *IEEE Comm. Surveys & Tutorials*, 14(4):1232–1243, 2012.

[17] P. Simoens et al. Vision: mapping the world in 3d through first-person vision devices with mercator. In *4th ACM Workshop on Mobile Cloud Computing and Services, Proc.*, pages 3–8. ACM, 2013.

[18] The OSGi Alliance. *OSGi Service Platform, Core Release 5*. aQute, 2012.

[19] The OSGi Alliance. *OSGi Service Platform, Enterprise Release 5*. aQute, 2012.

[20] B. Triggs et al. Bundle Adjustment - A Modern Synthesis. In *Vision Algorithms: Theory and Practice*, pages 298–372. Springer, 2000.

[21] B. Vanbrabant et al. A framework for integrated configuration management tools. In *IFIP/IEEE Int. Symp. on Integrated Network Management*, 2013.

[22] T. Verbelen et al. AIOLOS: middleware for improving mobile application performance through cyber foraging. *Journal Of Systems And Software*, 85(11):2629–2639, 2012.

[23] X. Zhang et al. Towards an Elastic Application Model for Augmenting the Computing Capabilities of Mobile Devices with Cloud Computing. *Mobile Networks and Applications*, 16(3):270–284, 2011.

[6] `https://cloud.google.com/products/app-engine`